ulm university universität
uulm

Institut für Angewandte Informationsverarbeitung
Dr. Andreas F. Borchert

16. Dezember 2014
Blatt 10

Objektorientierte Programmierung mit C++ (WS 2014/2015)

Abgabe bis zum 13. Januar 2015, 14:00 Uhr

Lernziele:

- Umgang mit Lambda-Ausdrücken in C++
- Rekursive Konstruktion von Funktionen mit Hilfe von Lambda-Ausdrücken

Aufgabe 13: Eine kleine Sprache

Zu implementieren ist eine kleine Sprache mit Ausdrücken, wobei für jeden eingelesenen Ausdruck ein Lambda-Ausdruck konstruiert wird, der, wenn er aufgerufen wird, diesen ausführt. Prinzipiell ist freigestellt, wie die kleine Sprache aussieht, aber sie sollte soweit gehen, dass sie die Definition einfacher Funktionen oder Variablen erlaubt.

Konkret vorgeschlagen wird folgende ein wenig an Lisp und Scheme anlehnende Grammatik, für die eine syntaktische Analyse als Ausgangsbasis zur Verfügung steht:

$$
\begin{aligned}
\langle \text{expr} \rangle &\longrightarrow \langle \text{identifier} \rangle \\
&\longrightarrow \langle \text{integer} \rangle \\
&\longrightarrow \text{„(“ } \langle \text{expr} \rangle \ \langle \text{expr} \rangle \text{ „)“} \\
&\longrightarrow \text{„(“ } \textbf{lambda} \ \langle \text{identifier} \rangle \ \langle \text{expr} \rangle \text{ „)“} \\
&\longrightarrow \text{„(“ } \textbf{define} \ \langle \text{identifier} \rangle \ \langle \text{expr} \rangle \text{ „)“} \\
&\longrightarrow \text{„(“ } \textbf{if} \ \langle \text{expr} \rangle \ \langle \text{expr} \rangle \ \langle \text{expr} \rangle \text{ „)“}
\end{aligned}
$$

Damit auch gewohnte Operatoren darstellbar sind, sind bei *identifier* auch diverse Sonderzeichen wie „+“, „-“ usw. zugelassen.

Der Wert eines Ausdrucks ist entweder eine ganze Zahl oder eine Funktion. Wenn im Interpreter das Resultat eine Zahl ist, wird diese ausgegeben, im Falle einer Funktion unterbleibt die Ausgabe:

```
> 77
77
> (lambda x x)
>
```

Der **lambda**-Operator entspricht hier dem $\lambda x.x$. Grundsätzlich müssen alle Operatoren mitsamt ihren Parametern eingeklammert werden.

Aus Gründen der Einfachheit haben hier alle selbstdefinierten Funktionen nur einen Parameter. Wenn dann beispielsweise ein Additions-Operator im Interpreter vordefiniert wird, dann erhält er nur den ersten Summanden und liefert dann eine Funktion, die den zweiten Summanden akzeptiert und dann die Summe zurückliefert. Das führt zu einer etwas umständlicheren Notation:

```
> ((+ 5) 6)
11
```

Mit Hilfe des **define**-Operators lassen sich Namen mit Ausdrücken fest verknüpfen:

```
> (define ten 10)
10
> ten
10
```

Die Kombination von **define** und **lambda** erlaubt die Definition benannter Funktionen:

```
> (define square (lambda x ((* x) x)))
> (square 7)
49
```

Zu beachten ist, dass bei Funktionsaufrufen hier der Parameter zuerst bewertet wird, bevor der Aufruf stattfindet. Wir haben somit traditionelles *call by value*, keine *normal-order evaluation*. (Letzteres wäre aber auch denkbar, macht die Implementierung aber etwas komplexer.)

Bezüglich der Sichtbarkeit von Namen gibt es zwei Namensräume. Jeder **lambda**-Ausdruck eröffnet einen lokalen lexikalischen Sichtbarkeitsbereich, in dem die Variable des Ausdrucks definiert ist (im obigen Beispiel ist das x). Dabei wird eine lexikalische Blockstruktur unterstützt. Im folgenden Beispiel ist in der inneren **lambda**-Definition sowohl x als auch y sichtbar:

```
> (define sum-of-squares
|    (lambda x (lambda y ((+ (square x)) (square y)))))
> ((sum-of-squares 3) 7)
58
```

Zusätzlich gibt es noch den globalen Namensraum mit den durch **define** definierten Namen. Das Durchsuchen des globalen Namensraumes erfolgt zur Laufzeit. Auf diese Weise sind rekursive Definitionen sehr leicht möglich:

```
> (define fak (lambda n (if ((= 0) n) 1 ((* n) (fak ((- n) 1))))))
> (fak 5)
120
```

Da der Parameter einer Funktion jeweils vor dem Aufruf bewertet wird, kann ein **if**-Operator nicht selbst gebaut werden. Der in der Grammatik definierte **if**-Operator wertet in Abhängigkeit des ersten Operanden nur den zweiten oder dritten Operanden aus.

Hinweise: Die oben definierte kleine Sprache besitzt zwei Datentypen: ganze Zahlen und Funktionen. Ein **lambda**-Operator liefert beispielsweise eine Funktion und keine Zahl. Bei Funktionen wird nicht nur ein Zeiger auf den sie implementierenden Lambda-Ausdruck benötigt, sondern auch die jeweilige Hülle (*closure*). Ein entsprechender Datentyp der beide Ausprägungen unterstützt, ist mit *Value* in *value.hpp* definiert.

Da mehrere Datentypen zueinander rekursiv sind, gibt es *types.hpp*, das mehrere Typen vordeklariert. Dort findet sich auch der Datentyp, mit dem Lambda-Ausdrücke verpackt werden können:

```
class Value;
typedef std::shared_ptr<Value> ValuePtr;

class Stack;
typedef std::shared_ptr<Stack> StackPtr;

typedef std::function<ValuePtr(StackPtr)> Function;
typedef std::shared_ptr<Function> FunctionPtr;
```

Stack dient der Parameterübergabe. Wenn mit **lambda** definierte Funktionen nur jeweils einen Parameter haben können, lassen sich diese recht einfach auf einem Stack finden, wobei mit festen Offsets gearbeitet werden kann. Alternativ wäre auch ein verschachtelbarer *Bindings*-Typ denkbar, der Namen jeweils Werte (also *ValuePtr*) zuordnet.

Die syntaktische Analyse (in *parser.cpp*) baut dann rekursiv Objekte des Typs *FunctionPtr*, die die erzeugten Lambda-Ausdrücke verpacken. Diese Funktionen wiederum benötigen jeweils die Parameter auf dem Stack (*StackPtr*) und liefern immer Objekte des Typs *ValuePtr*.

Grundsätzlich wird empfohlen, konsequent von intelligenten Zeigern (*std*::*shared_ptr*) Gebrauch zu machen, weil damit unerwünschte Duplikationen vermieden werden und die Freigabe automatisiert wird.

Wenn Sie die Vorlage verwenden wollen, finden Sie dort alles implementiert mit Ausnahme der Lambda-Ausdrücke in *parser.cpp*. Überall, wo dort *FIXME* im Kommentar steht, ist ein passender Lambda-Ausdruck einzufügen. Sie müssen aber nicht die Vorlage verwenden oder sich eng daran halten. Es steht Ihnen frei, alles zu ändern und zu experimentieren.

Wenn Sie bei den Lambda-Ausdrücken die *lambda-capture* definieren, kommen Sie normalerweise mit [=] recht weit, wenn Sie mit intelligenten Zeigern arbeiten. Für die Verwendung von Referenzen sollte hier kein Bedarf bestehen.

Mit dem Vorlesungsbeispiel *Expressions.tar.gz* vom 8. Dezember steht auch alternatives Ausgangsmaterial zur Verfügung. Dort wäre dann konsequent *ExpressionPtr* durch *FunctionPtr* zu ersetzen, die ebenfalls mit Lambda-Ausdrücken erzeugt werden. Diese Variante ist deutlich einfacher, weil dabei Funktionen als Werte nicht vorkommen und lokale Variablen wegfallen:

```
typedef int Value;

class SymbolTable;
typedef std::shared_ptr<SymbolTable> SymbolTablePtr;

typedef std::function<Value(SymbolTablePtr)> Function;
typedef std::shared_ptr<Function> FunctionPtr;
```

Wenn Sie die Lösung einreichen, sollten Sie sie zuvor mit *tar* verpacken:

```
thales$ tar cvf lambda.tar *.?pp Makefile
thales$ submit cpp 13 lambda.tar
```

Viel Erfolg!